Spring Data Elasticsearch

目录

README

> 因为工作需要学习Elasticsearch, 发现关于Spring-Data-Elasticsearch的文档非常少, 而且没有对应官方文档的中译版本，所以利用早起的空闲时间翻译一下。基于1.3.2.RELEASE版本，如果对Spring Data的其他模块(eg:Spring Data Jpa)比较了解，可以直接看第五章。

英文原版地址:Spring Data Elasticsearch

---

spring-data-elasticsearch 例子：github
spring-data-elasticsearch 与 Spring web结合 例子:github

下载地址：download
阅读地址：http://es.yemengying.com
博客地址：giraffe's home

参考文档

- http://www.ibm.com/developerworks/cn/opensource/os-cn-spring-jpa/
- http://leo-suen.iteye.com/blog/1991305

译者:giraffe0813

项目地址: github gitbook

关键字约定：

- domain class -> 域对象
- query builder -> 查询构造器

说明: 由于英文水平有限，且对querydsl等内容并不太了解，可能存在翻译有误的地方，请见谅，欢迎提issue

Created by giraffe0813 This is a book powered by GitBook.

前言

> Spring Data Elasticsearch为文档的存储，查询，排序和统计提供了一个高度抽象的模板。在使用中，你会发现Spring Data Elasticsearch和Spring Data Solr/Mongodb有许多相似之处。

项目元数据

- Version Control - https://github.com/spring-projects/spring-data-elasticsearch
- Bugtracker - https://jira.spring.io/browse/DATAES
- Release repository - https://repo.spring.io/libs-release
- Milestone repository - https://repo.spring.io/libs-milestone
- Snapshot repository - https://repo.spring.io/libs-snapshot

使用条件

需要使用Elasticsearch 0.20.2及以上的版本

玩转Spring Data Repositories

抽象出Spring Data repository是因为在开发过程中，常常会为了实现不同持久化存储的数据访问层而写大量的大同小异的代码。Spring Data repository的目的就是要大幅减少这些重复的代码。

> 本章阐述了Spring Data repository的核心概念及接口。本章的内容来自Spring data的公共模块，配置和样例代码来自Java Persistence Api(JPA)模块。如有需要，可以将XML文件的命名空间调整到你所使用的特定模块(eg:elasticsearch/mongo 等等)。命名空间参考文档涵盖了所有支持Repository API的Spring Data模块的XML配置。Repository查询关键字说明文档内列出了repository支持的查询方法的关键字。如果想要了解其他特定模块的详细信息，可以查询指定模块的参考文档。

核心概念

Spring Data repository抽象中最核心的接口就是*Repository*(显而易见的哈)。该接口使用了泛型，需要提供两个类型参数，第一个是接口处理的域对象类型，第二个是域对象的主键类型。这个接口常被看做是一个标记型接口，用来获取要操作的域对象类型和帮助开发者识别继承这个类的接口。在Repository的基础上，*CrudRepository*接口提供了针对实体类的复杂的CRUD(增删改查)操作。

Example 1.CrudRepository interface(CrudRepository接口)

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity); //1

    T findOne(ID primaryKey);       //2

    Iterable<T> findAll();          //3

    Long count();                   //4

    void delete(T entity);          //5

    boolean exists(ID primaryKey);  //6

    // … more functionality omitted.
}

1.保存实体类
2.返回指定id的实体类
3.返回所有实体类
4.返回实体类的数量
5.删除指定实体类
6.判断指定id的实体类是否存在
```

> Spring Data也提供了一些针对特定持久化技术的抽象，例如*JpaRepository*和*MongoRepository*。这些接口均继承了*CrudRepository*

*PagingAndSortingRepository*接口在*CrudRepository*的基础上增加了一些方法，使开发者可以方便的对实体类进行分页和排序。

Example 2.PagingAndSortingRepository

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
  extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

在分页长度为20的基础上，想要获取第二页的*User*数据，代码如下

```
PagingAndSortingRepository<User, Long> repository = // … get access to a bean
Page<User> users = repository.findAll(new PageRequest(1, 20));
```

除了查询方法，还可以使用在查询基础上衍生出的count(获取数量)和delete(删除)方法。

Example 3.Derived Count Query(获取数量)

```
public interface UserRepository extends CrudRepository<User, Long> {

  Long countByLastname(String lastname);
}
```

Example 3.Derived Delete Query(删除)

```
public interface UserRepository extends CrudRepository<User, Long> {

  Long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);

}
```

查询方法

标准的CRUD(增删改查)功能都要使用查询语句来查询数据库。但通过使用Spring Data，只要四个步骤就可以实现。

1.声明一个继承Repository接口或其子接口的持久层接口。并标明要处理的域对象类型及其主键的类型(在下面的例子中，要处理的域对象是Person，其主键类型是Long)

```
interface PersonRepository extends Repository<Person, Long> {...}
```

2.在接口中声明查询方法(spring会为其生成实现代码)

```
interface PersonRepository extends Repository<Person, Long> {
  List<Person> findByLastname(String lastname);
}
```

3.让Spring创建对这些接口的代理实例。
通过JavaConfig

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
class Config {}
```

通过XML配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/data/jpa
     http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

   <jpa:repositories base-package="com.acme.repositories"/>

</beans>
```

这里使用JPA的命名空间作为例子，需要根据实际使用的模块更改命名空间，比如可以改为mongodb等等。需要注意的是，在使用JavaConfig时，如果需要自定义扫描的包而不是使用其默认值，可以利用注解@EnableJpaRepositories的basePackage属性。具体使用方式如下:

```
@EnableJpaRepositories(basePackages = "com.cshtong")//单个包
@EnableJpaRepositories(basePackages = {"com.cshtong.sample.repository", "com.cshtong.towe
```

4.注入repository实例，并使用

```
public class SomeClient {

  @Autowired
  private PersonRepository repository;

  public void doSomething() {
    List<Person> persons = repository.findByLastname("Matthews");
  }
}
```

这一部分会在之后详细解释

repository接口定义

在第一步中我们定义了一个针对特定域对象的repository接口，接口继承了Repository接口并且标明了域对象类型及其主键类型。如果想要暴露CRUD方法可以不继承Repository接口，直接继承CrudRepository接口即可。

调整repository定义

通常情况下，我们的repository接口会继承Repository,CrudRepository或PagingAndSortingRepository接口。但是，如果不想通过extend关键字继承Spring Data的接口，还可以采用在自定义的repository接口上加注解的方式，两种方式是等价的。例子如下：

```
public interface UserDao extends Repository<AccountInfo, Long>{...}

@RepositoryDefinition(domainClass = Person, idClass = Long.class)
public interface PersonRepository {...}
```

此外，继承CrudRepository接口，会为操作域对象提供了一组可立即使用的方法。如果开发者不想暴露CrudRepository接口的全部方法，只需简单的将需要的方法复制到自己的repository接口中。

> 这让我们可以在Spring Datade Repository的基础上定义自己的抽象接口

Example 5.Selectively exposing CRUD methods(选择性暴露方法)

```
@NoRepositoryBean
interface MyBaseRepository<T, ID extends Serializable> extends Repository<T, ID> {

  T findOne(ID id);

  T save(T entity);
}

interface UserRepository extends MyBaseRepository<User, Long> {

  User findByEmailAddress(EmailAddress emailAddress);
}
```

在上面的代码中，我们先定义了一个公共的基础接口MyBaseRepository，并提供findOne和save两个方法。接着声明一个UserRepository的接口并继承基础接口MyBaseRepository，所以UserRepository拥有保存User，根据id查找user和根据邮箱地址查找User三个方法。

> 注意，中间过渡的接口MyBaseRepository上加了NoRepositoryBean注解，这个注解是告诉Spring Data不要为这个接口创建repository代理实例。

定义查询方法

通过方法名，Spring Data有两种方式获得开发者的查询意图。一是直接解析方法名，二是使用@Query创建的查询。那么到底使用哪种方式呢？Spring Data有一套策略来决定创建的查询。

查询策略

Spring Data repository使用下面的一套策略来决定最后创建的查询。可以通过配置XML中的 *query-loop-strategy*属性或Javaconfig中Enable${store}Repositories注解的 *queryLookupStrategy*属性来调整策略。某些特定的数据存储可能不支持所有策略。

- *CREATE* 通过解析方法名构建查询，会删除方法名的某些前缀(eg:findBy)，并解析剩下的部分，会在下一节创建查询中详细讲。
- *USE_DECLARED_QUERY* 如果方法通过 @Query 指定了查询语句，则使用该语句创建Query；如果没有，则查找是否定义了符合条件的Named Query，如果找到，则使用该命名查询；如果两者都没有找到，则抛出异常。使用@Query声明查询语句的例子如下:

```
//使用Query注解
@Query ("select a from AccountInfo a where a.accountId = ?1")
public AccountInfo findByAccountId (Long accountId) ;
```

- *CREATE_IF_NOT_FOUND*(默认) 结合了*CREATE*和*USE_DECLARED_QUERY* 两种策略,会先尝试查找声明好的查询，如果没有找到，就按照解析方法名的方式构建查询。这是默认的查询策略，如果不更改配置，会一直使用这种策略构建查询。这种策略支持通过方法名快速定义一个查询，也允许引入声明好的查询。

创建查询

Spring Data repository自带了一个非常有用的查询构造器。它会从方法名中去掉类似*find..By*,*read...By*,*query...By*,*count...By*之类的前缀，然后解析剩余的名字。我们也可以在方法名中加入更多的表达式，比如查询时需要distinct约束,那么在方法名中加入Distinct即可。方法名中的第一个By是一个分解符，代表着查询语句的开始，我们可以用And或Or来将多个查询条件关联起来。

Example 6.Query creation from method names(通过方法名创建查询)

```
public interface PersonRepository extends Repository<User, Long> {

  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

  // Enables the distinct flag for the query
  // 在查询中使用distinct约束
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname)
  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname)

  // Enabling ignoring case for an individual property
  // 忽略大小写 在方法名中加入IgnoreCase
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling ignoring case for all suitable properties
  // 所有属性都忽略大小写 在方法名中加入AllIgnoreCase
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname)

  // Enabling static ORDER BY for a query
  // 排序 在方法名中加入OrderBy
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

对方法名的解析结果取决于实际要操作的数据库/搜索引擎是什么。另外，还有一些普遍的问题要注意:

- 在方法名中，可以使用*And*和*Or*连接多个属性。除此之外还可以使用*Between*,*LessThan*,*GreaterThan*,*Like*等等，不同数据库支持的连接符是不一样的，需要查看相关文档
- 可以使用IgnoreCase来忽略单个属性的大小写，比如*findByLastnameIgoreCase*,也可以使用*AllIgnoreCase*来忽略全部属性的大小写。前提是，实际选择的数据库/搜索引擎支持。
- 可以使用*OrderBy*来对相关属性进行排序。具体是升序还是降序，是通过*Asc*和*Desc*控制的。如果想了解动态排序，请看处理特殊参数。

属性表达式

接下来我们来看看属性表达式。在之前的例子中，属性表达式只涉及到被管理的实体类的直接属性，在创建查询时我们已经确保解析出的属性是被管理实体类的属性之一。实际上，我们可以定义嵌套属性。假设Person类有一个Address，Address中又有ZipCode。在这种情况下，下面方法的方法名会通过x.address.zipCode来检索属性。

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

Spring Data的查询构造器会先把*AddressZipCode*当做一个整体，检查其是不是实体类的一个属性。如果是，就使用这个属性。如果不是，会按照驼峰式从右向左进行分割，再进行属性匹配。在上面的例子中，会将*AddressZipCode*分为*AddressZip*和*Code*。如果第一次分割之后还没有找到相匹配的属性，构造器会左移分割点重新进行分割，分为*Address*和*ZipCode*，并继续解析是否有相匹配的属性。 在大多数情况下，构造器都能够正确的解析出相匹配的属性。但在某些特殊情况下，可能会选择了错误的属性。假设*Person*类中除了*address*外，还有一个*addressZip*属性。那么构造器会在第一次分割后就匹配到相应的属性，然后报错（因为*addressZip*中没有*code*属性）。 为了避免的解析的歧义问题，我们可以在方法名中使用'_'符号来显式的表达意图，更改后的方法名如下:

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

因为构造器已经将下划线作为保留字符，所以强烈建议开发者遵循标准的Java命名规范，不要在属性名中使用下划线，采用驼峰式命名。

处理特殊参数

通过上面的例子，我们知道可以方便的通过定义方法的参数来处理查询中的参数。除此之外，我们还可以为方法添加某些特定类型的参数(如:*Pageable*和*Sort*)来动态的在查询中添加分页和排序。

Example 7. Using Pageable, Slice and Sort in query methods(查询中进行分页和排序)

```
Page<User> findByLastname(String lastname, Pageable pageable);

Slice<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
```

第一个例子中我们向方法传递一个*org.springframework.data.domain.Pageable*的实例来为查询动态的添加分页，返回的*Page*对象中包含元素总数和当前页的数据。其中的元素总数是通过Spring Data自动触发的一个count查询获得的。但是count查询有时会降低一定的性能，所以在不需要总数时，我们可以使用*Slice*来代替*Page*。Slice仅仅可以知道是否有可用的下一个Slice。 排序操作也可以通过Pageable实例来处理。但如果你需要的只是排序，只需要为方法添加*org.springframework.data.domain.Sort*类型的参数即可。我们也可以只简单的返回一个list， 这时就不会执行count查询，只查询给定范围的实体类。

> 如果想知道总共有多少页，就必须触发一个额外的count查询

限定查询结果集大小

Spring Data允许开发者使用*first*和*top*关键字对返回的查询结果集大小进行限定。fisrt和top需要跟着一个代表返回的结果集的最大大小的数字。如果没有跟着一个数字，那么返回的结果集大小默认为1。

Example 8.Limiting the result size of query with Top and First(利用first和top限制返回的结果集大小)

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

限制结果集的表达式还支持*Distinct*关键字。并且，当返回的结果集大小限制为1时，Spring Data支持将返回结果包装到Optional(java 8新增，这是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象)之中，例子如下:

```
Optional<User> findFirstByOrderByLastnameAsc();
```

在查询用page和slice来进行分页查询的情况下，同样可以使用*first*和*top*来对结果集大小进行限制。

> 注意，如果在使用*Sort*参数对查询结果进行排序的基础上加上对结果集大小的限制，就可以轻易的获得最大的K个元素或最小的K个元素。

利用Stream(流)处理查询结果

我们可以使用Stream作为返回类型可以对查询结果进行逐个处理。

Example 9. Stream the result of a query with Java 8 Stream(stream查询结果)

```
@Query("select u from User u")
Stream<User> findAllByCustomQueryAndStream();

Stream<User> readAllByFirstnameNotNull();

@Query("select u from User u")
Stream<User> streamAllPaged(Pageable pageable);
```

> 一个Stream中可能包含底层数据存储的特定资源，所以在使用后必须关闭。可以通过调用close()方法，也可以使用java 7中的try-with-resources块

Example 10. Working with a Stream result in a try-with-resources block(在块中使用Stream)

```
try (Stream<User> stream = repository.findAllByCustomQueryAndStream()) {
  stream.forEach(…);
}
```

> 目前，不是所有的Spring Data模块都支持将Stream作为返回类型

异步处理查询结果

Spring Data repository中的查询可以异步执行，参考Spring执行异步方法。这意味着方法可以在被调用时立刻返回，而真正的查询执行会被当做一个任务提交到Spring的TaskExecutor。

```
@Async
Future<User> findByFirstname(String firstname); //1

@Async
CompletableFuture<User> findOneByFirstname(String firstname); //2

@Async
ListenableFuture<User> findOneByLastname(String lastname); //3

1.使用java.util.concurrent.Future作为返回类型。
2.使用Java 8中的java.util.concurrent.CompletableFuture作为返回类型。
3.使用org.springframework.util.concurrent.ListenableFuture作为返回类型
```

创建repository实例

这一部分将介绍如何为定义好的repository接口创建实例和bean。一种方式是使用Spring Data模块下支持repository配置的Spring命名空间。不过，通常推荐使用JavaConfig风格的配置。

XML配置

每个Spring Data模块都包含一个repository元素，通过这个元素开发者可以轻松的定义spring扫描包的路径。

Example 11.Enabling Spring Data repositories via XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <repositories base-package="com.acme.repositories" />

</beans:beans>
```

在上面的例子中，Spring扫描base-package中指定的路径及其包含的所有包，检测出继承Repository或者其子接口的接口。每找到一个接口，FactoryBean会创建对应的代理去处理查询方法的调用。每个注册的bean的名称都来自于接口名称，例如：UserRepository将会被注册为userRepository。base-package允许使用通配符来定义扫描路径。

使用过滤器

默认情况下，Spring会找到配置路径下的所有接口并为其创建一个bean实例。但有些情况下开发者可能想要更细致的控制创建bean实例的接口。这时候，可以在元素中使用和。这些元素的详细用法可以参考spring参考文档。 下面的配置就是排除某些特定接口的例子:

Example 12. Using exclude-filter element(使用exclude-filter元素)

```
<repositories base-package="com.acme.repositories">
  <context:exclude-filter type="regex" expression=".*SomeRepository" />
</repositories>
```

上面的配置排除了所有以SomeRepository结尾的接口。

JavaConfig

我们也可以使用JavaConfig类中的注解@Enable${store}Repositories来配置扫描包的路径。在实际开发中，要针对不同的持久化存储替换不同的${store}。下面是使用JavaConfig配置的一个例子: Example 13. Sample annotation based repository configuration

```
@Configuration
@EnableJpaRepositories("com.acme.repositories")
class ApplicationConfiguration {

  @Bean
  public EntityManagerFactory entityManagerFactory() {
    //...
  }
}
```

> 在上面的例子中使用了针对Jpa的注解，需要根据实际使用的存储修改。同样地，使用的存储不同，定义的*EntityManagerFactory*bean也不同。具体请查看针对指定存储配置的文档。

独立使用

开发者可以在脱离Spring容器的情况下利用RepositoryFactory来使用Spring Data repository(比如在CDI环境下)，但仍然需要将某些Spring的依赖包加到classpath中。

Example 14. Standalone usage of repository factory（独立使用）

```
RepositoryFactorySupport factory = … // Instantiate factory here(初始化factory)
UserRepository repository = factory.getRepository(UserRepository.class);
```

自定义repository实现

在开发过程中，常常需要为一些repository方法添加自定义的实现。Spring Data repository允许开发者自定义repository方法。

为单一的repositories添加自定义方法

为了通过自定义方法来丰富repository，首先要定义一个接口和一个接口的实现类。

Example 15. Interface for custom repository functionality(自定义接口)

```
interface UserRepositoryCustom {
  public void someCustomMethod(User user);
}
```

Example 16. Implementation of custom repository functionality(接口的实现类)

```
class UserRepositoryImpl implements UserRepositoryCustom {

  public void someCustomMethod(User user) {
    // Your custom implementation
  }
}
```

> 接口的实现类名字后缀必须为*impl*才能在扫描包时被找到。

实现类本身并不依赖于Spring Data，就是一个普通的Srping的bean，所以可以是用标准的依赖注入在其中注入其它相关的bean，比如JdbcTemplate等等。

Example 17. Changes to the your basic repository interface(修改基础的repository接口)

```
interface UserRepository extends CrudRepository<User, Long>, UserRepositoryCustom {

  // Declare query methods here
}
```

上面代码在让标准的repository接口继承了自定义的接口，将标准的CRUD和自定义的方法结合到了一起。

配置

在使用Xml配置时，Spring框架会自动检测指定包路径下的实现类。实现类的后缀必须满足属性*repository-impl-postfix*的定义，默认后缀是*Impl*。

```
<repositories base-package="com.acme.repository" />

<repositories base-package="com.acme.repository" repository-impl-postfix="FooBar" />
```

在第一个配置中，spring会找到类*com.acme.repository.UserRepositoryImpl*，而在第二个配置中，spring会尝试查找*com.acme.repository.UserRepositoryFooBar*类

人工装载

上面的例子使用了基于注解的配置来自动装载。如果自定义的实现类需要特殊的装载，也可以手动的声明一个bean,名字要遵循上一节定义的命名规范。

Example 19. Manual wiring of custom implementations(人工装载自定义的实现类)

```
<repositories base-package="com.acme.repository" />
<beans:bean id="userRepositoryImpl" class="...">

  <!--其他配置 -->
</beans:bean
```

为所有的repositories添加自定义方法

如果想要给为所有的repository接口增加一个方法，那么之前的方法是不可行的。为了给所有的repository添加自定义的方法，开发者首先需要定义一个中间过渡的接口。

Example 20. An interface declaring custom shared behavior(定义中间接口声明)

```
@NoRepositoryBean
public interface MyRepository<T, ID extends Serializable>
  extends PagingAndSortingRepository<T, ID> {

  void sharedCustomMethod(ID id);
}
```

现在，开发者的其他repository接口需要继承定义的中间接口而不是继承*Repository*接口。接下来需要为这个中间接口创建实现类。这个实现类是基于Repository中的基类的(不同的持久化存储继承的基类也不相同，这里以Jpa为例)，这个类会被当做repository代理的自定义类来执行。 Example 21. Custom repository base class(自定义reposotory基类)

```
public class MyRepositoryImpl<T, ID extends Serializable>
  extends SimpleJpaRepository<T, ID> implements MyRepository<T, ID> {

  private final EntityManager entityManager;

  public MyRepositoryImpl(JpaEntityInformation entityInformation,EntityManager entityMana
    super(entityInformation, entityManager);

    // Keep the EntityManager around to used from the newly introduced methods.
    this.entityManager = entityManager;
  }

  public void sharedCustomMethod(ID id) {
    // implementation goes here
  }
}
```

Spring中命名空间默认会为base-package下所有的接口提供一个实例。但MyRepository只是作为一个中间接口，并不需要创建实例。所以我们可以使用@NoRepositoryBean注解或将其排除在base-package的配置中

最后一步是使用JavaConfig或Xml配置这个自定义的repository基类。

Example 22. Configuring a custom repository base class using JavaConfig(使用JavaConfig配置repository基类)

```
@Configuration
@EnableJpaRepositories(repositoryBaseClass = MyRepositoryImpl.class)
class ApplicationConfiguration { … }
```

也可使用Xml的方式配置。

Example 23. Configuring a custom repository base class using XML(使用xml配置自定义repository基类)

```
<repositories base-package="com.acme.repository"
     repository-base-class="….MyRepositoryImpl" />
```

扩展Spring Data

这一章将介绍如何把Spring Data扩展到其他的框架中。接下来让我们看看如何将Spring Data整合到Spring MVC中。

WEB支持

Spring Data带有很多的web支持。要使用它们，需要在classpath中加入Spring MVC的jar包，有的还需要整合Spring HATEOAS。通常情况下，只需在JavaConfig的配置类中使用*@EnableSpringDataWebSupport*注解即可。

Example 24. Enabling Spring Data web support(使用Spring Data的web支持)

```
@Configuration
@EnableWebMvc
@EnableSpringDataWebSupport
class WebConfiguration { }
```

*@EnableSpringDataWebSupport*注解会注册一些组件，我们会在之后讨论。它同样也会去检测classpath中的Spring HATEOAS,并且注册他们。

如果不想通过JavaConfig开启web支持，也可以使用xml配置，将*SpringDataWebSupport*和*HateoasAwareSpringDataWebSupport*注册为Spring的bean。

Example 25. Enabling Spring Data web support in XML(使用xml配置)

```
<bean class="org.springframework.data.web.config.SpringDataWebConfiguration" />

<!-- If you're using Spring HATEOAS as well register this one *instead* of the former -->
<bean class="org.springframework.data.web.config.HateoasAwareSpringDataWebConfiguration"
```

基础的web支持

上面的配置会注册一些基础组件:

- *DomainClassConverter*:使Spring MVC可以从请求参数或路径变量中解析repository管理的域对象的实例。
- *HandlerMethodArgumentResolver*:使Spring mvc能从请求参数来解析Pageable和Sort实例

DomainClassConverter

DomainClassConverter 允许开发者在SpringMVC控制层的方法中直接使用域对象类型(Domain types),而无需通过repository手动查找这个实例。

Example 26. A Spring MVC controller using domain types in method signatures (在Spring MVC控制层方法中直接使用域对象类型)

```
@Controller
@RequestMapping("/users")
public class UserController {

  @RequestMapping("/{id}")
  public String showUserForm(@PathVariable("id") User user, Model model) {

    model.addAttribute("user", user);
    return "userForm";
  }
}
```

上面的方法直接接收了一个User对象,开发者不需要做任何的搜索操作,转换器会自动将路径变量id转为User对象的id,并且调用了findOne()方法查询出User实体。

> 注意:当前的Repository 必须实现CrudRepository

HandlerMethodArgumentResolver

这个配置同时注册了*PageableHandlerMethodArgumentResolver*和*SortHandlerMethodArgumentResolver*,是开发者可以在controller的方法中使用*Pageable*和*Sort*作为参数。 Example 27. Using Pageable as controller method argument（在controller中使用Pageable作为参数）

```
@Controller
@RequestMapping("/users")
public class UserController {

  @Autowired UserRepository repository;

  @RequestMapping
  public String showUsers(Model model, Pageable pageable) {

    model.addAttribute("users", repository.findAll(pageable));
    return "users";
  }
}
```

通过上面的方法定义，Spring MVC会使用下面的默认配置尝试从请求参数中得到一个Pageable的实例。 Table 1. Request parameters evaluated for Pageable instances page(由于构造Pageable实例的请求参数)

参数名	作用
page	想要获取的页数，默认为0
size	获取页的大小，默认为20
sort	需要排序的属性，格式为*property,property(,ASC/DESC)*,默认升序排序。支持多个字段排序，比如*?sort=firstname&sort=lastname,asc*

如果开发者想要自定义分页或排序的行为，可以继承*SpringDataWebConfiguration*或*HATEOAS-enabled*，并重写*pageableResolver()*或*sortResolver()*方法，引入自定义的配置文件来代替使用@Enable-注解。

开发者也可以针对多个表定义多个*Pageable*或*Sort*实例,需要使用Spring的@*Qualifier*注解来区分它们。并且请求参数名要带有*${qualifier}_*的前缀。例子如下:

```
public String showUsers(Model model,
      @Qualifier("foo") Pageable first,
      @Qualifier("bar") Pageable second) {...}
```

请求中需要带有foo_page和bar_page等参数。

默认的*Pageable*相当于*new PageRequest(0, 20)*，但开发者可以在*Pageable*参数上使用@*PageableDefaults*来自定义。

超媒体分页

Spring HATEOAS有一个PagedResources类,它丰富了Page实体以及一些让用户更容易导航到资源的链接。Page转换到PagedResources是由一个实现了Spring HATEOAS ResourceAssembler接口的实现类:PagedResourcesAssembler提供转换的。 Example 28. Using a PagedResourcesAssembler as controller method argument(使用PagedResourcesAssembler当做方法参数)

```
@Controller
class PersonController {

  @Autowired PersonRepository repository;

  @RequestMapping(value = "/persons", method = RequestMethod.GET)
  HttpEntity<PagedResources<Person>> persons(Pageable pageable,
    PagedResourcesAssembler assembler) {

    Page<Person> persons = repository.findAll(pageable);
    return new ResponseEntity<>(assembler.toResources(persons), HttpStatus.OK);
  }
}
```

上面的toResources方法会执行以下的几个步骤:

- Page对象的content会转换成为PagedResources对象的content。
- PagedResources会的到一个PageMetadata的实体,包含从Page跟PageRequest得到的信息。
- PagedResources会根据状态得到prev跟next链接,这些链接指向URI所匹配的方法。分页参数会根据PageableHandlerMethodArgumentResolver配置,以让其能够在后面的方法中解析使用。

假设我们数据库中存有30个人。发送一个GET请求*http://localhost:8080/persons*,得到的返回结果如下：

```
{ "links" : [ { "rel" : "next",
                "href" : "http://localhost:8080/persons?page=1&size=20 }
  ],
  "content" : [
     … // 20 Person instances rendered here
  ],
  "pageMetadata" : {
    "size" : 20,
    "totalElements" : 30,
    "totalPages" : 2,
    "number" : 0
  }
}
```

从返回结果中可以看出assembler生成了正确的URI，并根据默认配置设置了分页的请求参数。这意味着，如果我们更改了配置，这个链接会自动更改。默认情况下，assembler生成的链接会指向被调用的controller方法，但也可以通过重写*PageResourceAssembler.toResource{...}*方法提供一个自定义的链接。

QueryDSL web支持

对于那些集成了QueryDSL的存储可以从请求的查询参数中直接获得查询语句。 这意味着下面的针对*User*的请求参数:

```
?firstname=Dave&lastname=Matthews
```

会通过*QuerydslPredicateArgumentResolver*解析成:

```
QUser.user.firstname.eq("Dave").and(QUser.user.lastname.eq("Matthews"))
```

> 如果使用了@EnableSpringDataWebSupport注解，并且classpath中包含Querydsl，那么该功能会自动开启。

在方法中加入*@QuerydslPredicate*注解，可以提供我们使用*Predicate*

```
@Controller
class UserController {

  @Autowired UserRepository repository;

  @RequestMapping(value = "/", method = RequestMethod.GET)
  String index(Model model, @QuerydslPredicate(root = User.class) Predicate predicate,
          Pageable pageable, @RequestParam MultiValueMap<String, String> parameters) {

    model.addAttribute("users", repository.findAll(predicate, pageable));

    return "index";
  }
}
1.解析查询参数来匹配针对user的Predicate
```

默认的绑定如下:

- 一个对象对应一个简单属性相当于*eq*

  ```
  ?firstname=2
  ```

- 多个属性上对应一个对象相当于*contains*

  ```
  ?firstname=2&firstname=3
  ```

- 简单属性对应一个集合相当于*in*

  ```
  ?firstname=[2,3,4,5]
  ```

这些绑定规则可以通过*@QuerydslPredicate*的*bindings*属性或者使用Java 8新引入的*default*方法在repository接口中加入*QuerydslBinderCustomizer*方法来更改。

```
interface UserRepository extends CrudRepository<User, String>,
                                 QueryDslPredicateExecutor<User>,       //1
                                  QuerydslBinderCustomizer<QUser> {      //2

  @Override
  default public void customize(QuerydslBindings bindings, QUser user) {

    bindings.bind(user.username).first ((path, value) -> path.contains(value))          /
    bindings.bind(String.class)
      .first((StringPath path, String value) -> path.containsIgnoreCase(value));        /
    bindings.excluding(user.password);  //5
  }
}

1.QueryDslPredicateExecutor provides access to specific finder methods for Predicate.
2.QuerydslBinderCustomizer defined on the repository interface will be automatically pick
3.Define the binding for the username property to be a simple contains binding.
4.Define the default binding for String properties to be a case insensitive contains matc
5.Exclude the password property from Predicate resolution.
```

Repository填充

如果使用过Spring提供的JDBC模块，可能会对使用SQL脚本填充*DataSource*。Spring Data中也提供了相似的功能来填充repository，只不过不是使用SQL，而是使用XML或JSON来定义数据，

假设有一个*data.json*文件，内容如下:

Example 29. Data defined in JSON（使用JSON定义数据）

```
[ { "_class" : "com.acme.Person",
 "firstname" : "Dave",
  "lastname" : "Matthews" },
  { "_class" : "com.acme.Person",
 "firstname" : "Carter",
  "lastname" : "Beauford" } ]
```

为了把前面定义的数据填充到PersonRepository中可以使用repository命名空间提供的popilator元素。 Example 30. Declaring a Jackson repository populator(声明Jackson repository populator)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:repository="http://www.springframework.org/schema/data/repository"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    http://www.springframework.org/schema/data/repository/spring-repository.xsd">

  <repository:jackson2-populator locations="classpath:data.json" />

</beans>
```

上面的配置会让*data.json*文件可以在其他地方通过Jackson的Object Mapper读取和反序列化。

JSON object的类型可以通过解析json文件中的_*class*属性得到。Spring框架会根据反序列化的对象选择合适的repository来处理。

如果想要用XML来定义repositries填充的数据，需要使用unmarshaller-populator元素,可以利用Spring OXM提供的组件，想要详细了解请参考Spring参考文档:

Example 31. Declaring an unmarshalling repository populator (using JAXB)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:repository="http://www.springframework.org/schema/data/repository"
  xmlns:oxm="http://www.springframework.org/schema/oxm"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    http://www.springframework.org/schema/data/repository/spring-repository.xsd
    http://www.springframework.org/schema/oxm
    http://www.springframework.org/schema/oxm/spring-oxm.xsd">

  <repository:unmarshaller-populator locations="classpath:data.json"
    unmarshaller-ref="unmarshaller" />

  <oxm:jaxb2-marshaller contextPath="com.acme" />

</beans>
```

Elasticsearch Repositories

这一章会详细介绍Elasticsearch repository实现。

介绍

Spring命名空间

Spring Data Elasticsearch模块包含一个自定义的命名空间，它允许我们定义repository bean和初始化一个*ElasticsearchServer*。

下面，我们像创建Repository实例中描述的那样使用*repositories*元素查找Spring Data repository。

Example 32. Setting up Elasticsearch repositories using Namespace(使用命名空间创建Elasticsearch repositories)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd">

    <elasticsearch:repositories base-package="com.acme.repositories" />

</beans>
```

使用*Transport Client*和*Node Client*元素注册一个*Elasticsearch Server*实例。

Example 33. Transport Client using Namespace(使用Transport Client)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd">

    <elasticsearch:transport-client id="client" cluster-nodes="localhost:9300,someip:9300

</beans>
```

Example 34. Node Client using Namespace(使用Node Client)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd">

    <elasticsearch:node-client id="client" local="true"" />

</beans>
```

基于注解的配置

Spring Data Elasticsearch repository可以通过XML配置，也可以通过JavaConfig的注解配置。

Example 35. Spring Data Elasticsearch repositories using JavaConfig(使用JavaConfig)

```
@Configuration
@EnableElasticsearchRepositories(basePackages = "org/springframework/data/elasticsearch/r
static class Config {

    @Bean
    public ElasticsearchOperations elasticsearchTemplate() {
        return new ElasticsearchTemplate(nodeBuilder().local(true).node().client());
    }
}
```

上面的配置使用*ElasticsearchTemplate*创建了一个*Embedded Elasticsearch Server*。通过*@EnableElasticsearchRepositories*注解来启动Spring Data Elasticsearch Repositories，如果没有显式指定扫描的包路径，会扫描配置类所在的包。

使用CDI

Spring Data Elasticsearch repositories也可以使用CDI注入。

Example 36. Spring Data Elasticsearch repositories using JavaConfig（使用JavaConfig）

```
class ElasticsearchTemplateProducer {

    @Produces
    @ApplicationScoped
    public ElasticsearchOperations createElasticsearchTemplate() {
        return new ElasticsearchTemplate(nodeBuilder().local(true).node().client());
    }
}

class ProductService {

    private ProductRepository repository;

    public Page<Product> findAvailableBookByName(String name, Pageable pageable) {
        return repository.findByAvailableTrueAndNameStartingWith(name, pageable);
    }

    @Inject
    public void setRepository(ProductRepository repository) {
        this.repository = repository;
    }
}
```

查询方法

查询策略

Elasticsearch模块支持所有基本查询构建，比如String,Abstract,Criteria或通过方法名获得构建查询。

声明查询

通过解析方法名来构建查询有时可能满足不了开发者的需求，或造成方法名可读性差。这时可以使用@*Query*注解来声明一个查询(参考使用@Query注解)

创建查询

通常情况下,Elasticsearch模块创建查询的机制与4.2节查询方法中描述的一样。通过下面的例子，我们来看看Elasticsearch模块会根据一个查询方法生成怎样的查询语句。

Example 37. Query creation from method names（通过方法名创建查询）

```
public interface BookRepository extends Repository<Book, String>
{
    List<Book> findByNameAndPrice(String name, Integer price);
}
```

根据上面的方法名会生成下面的Elasticsearch查询语句

```
{ "bool" :
    { "must" :
        [
            { "field" : {"name" : "?"} },
            { "field" : {"price" : "?"} }
        ]
    }
}
```

下面的表格列出了所有Elasticsearch支持的关键字。

Table 2. Supported keywords inside method names（方法名中支持的关键字）

关键字	例子	Elasticsearch查询语句
And	findByNameAndPrice	{"bool" : {"must" : [{"field" : {"name" : "?"}}, {"field" : {"price" : "?"}}]}}
Or	findByNameOrPrice	{"bool" : {"should" : [{"field" : {"name" : "?"}}, {"field" : {"price" : "?"}}]}}
Is	findByName	{"bool" : {"must" : {"field" : {"name" : "?"}}}}
Not	findByNameNot	{"bool" : {"must_not" : {"field" : {"name" : "?"}}}}

LessThanEqual	findByPriceLessThan	{"bool" : {"must" : {"range" : {"price" : {"from" : null,"to" : ?,"include_lower" : true,"include_upper" : true}}}}}
GreaterThanEqual	findByPriceGreaterThan	{"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : null,"include_lower" : true,"include_upper" : true}}}}}
Before	findByPriceBefore	{"bool" : {"must" : {"range" : {"price" : {"from" : null,"to" : ?,"include_lower" : true,"include_upper" : true}}}}}
After	findByPriceAfter	{"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : null,"include_lower" : true,"include_upper" : true}}}}}
Like	findByNameLike	{"bool" : {"must" : {"field" : {"name" : {"query" : "?*","analyze_wildcard" : true}}}}}
StartingWith	findByNameStartingWith	{"bool" : {"must" : {"field" : {"name" : {"query" : "?*","analyze_wildcard" : true}}}}}
EndingWith	findByNameEndingWith	{"bool" : {"must" : {"field" : {"name" : {"query" : "*?","analyze_wildcard" : true}}}}}
Contains/Containing	findByNameContaining	{"bool" : {"must" : {"field" : {"name" : {"query" : "?","analyze_wildcard" : true}}}}}
In	findByNameIn(Collectionnames)	{"bool" : {"must" : {"bool" : {"should" : [{"field" : {"name" : "?"}},

		{"field" : {"name" : "?"}}]}}}}
NotIn	findByNameNotIn(Collectionnames)	{"bool" : {"must_not" : {"bool" : {"should" : {"field" : {"name" : "?"}}}}}}
Near	findByStoreNear	暂不支持
True	findByAvailableTrue	{"bool" : {"must" : {"field" : {"available" : true}}}}
False	findByAvailableFalse	{"bool" : {"must" : {"field" : {"available" : false}}}}
OrderBy	findByAvailableTrueOrderByNameDesc	{"sort" : [{ "name" : {"order" : "desc"} }],"bool" : {"must" : {"field" : {"available" : true}}}}

使用@Query注解

Example 38. Declare query at the method using the @Query annotation.(使用@Query注解声明查询)

```
public interface BookRepository extends ElasticsearchRepository<Book, String> {
    @Query("{"bool" : {"must" : {"field" : {"name" : "?0"}}}}")
    Page<Book> findByName(String name,Pageable pageable);
}
```

其他Elasticsearch操作的支持

这一章会讲解无法通过repository接口直接使用的其他Elasticsearch操作。建议像自定义repository实现这章中描述的那样为repository添加自定义的实现。

构建Filter

使用过滤器可以提高查询速度

```
private ElasticsearchTemplate elasticsearchTemplate;

SearchQuery searchQuery = new NativeSearchQueryBuilder()
    .withQuery(matchAllQuery())
    .withFilter(boolFilter().must(termFilter("id", documentId)))
    .build();

Page<SampleEntity> sampleEntities =
    elasticsearchTemplate.queryForPage(searchQuery,SampleEntity.class);
```

利用Scan和Scroll处理大结果集

Elasticsearch在处理大结果集时可以使用scan和scroll。在Spring Data Elasticsearch中，可以向下面那样使用*ElasticsearchTemplate*来使用scan和scroll处理大结果集。

Example 39. Using Scan and Scroll(使用scan和scroll)

```
SearchQuery searchQuery = new NativeSearchQueryBuilder()
    .withQuery(matchAllQuery())
    .withIndices("test-index")
    .withTypes("test-type")
    .withPageable(new PageRequest(0,1))
    .build();
String scrollId = elasticsearchTemplate.scan(searchQuery,1000,false);
List<SampleEntity> sampleEntities = new ArrayList<SampleEntity>();
boolean hasRecords = true;
while (hasRecords){
    Page<SampleEntity> page = elasticsearchTemplate.scroll(scrollId, 5000L , new ResultsM
    {
        @Override
        public Page<SampleEntity> mapResults(SearchResponse response) {
            List<SampleEntity> chunk = new ArrayList<SampleEntity>();
            for(SearchHit searchHit : response.getHits()){
                if(response.getHits().getHits().length <= 0) {
                    return null;
                }
                SampleEntity user = new SampleEntity();
                user.setId(searchHit.getId());
                user.setMessage((String)searchHit.getSource().get("message"));
                chunk.add(user);
            }
            return new PageImpl<SampleEntity>(chunk);
        }
    });
    if(page != null) {
        sampleEntities.addAll(page.getContent());
        hasRecords = page.hasNextPage();
    }
    else{
        hasRecords = false;
    }
    }
}
```

附录

附录A 命名空间参考

元素

元素最重要的属性就是base-package,用来定义查找Spring Data repository接口时扫描包的路径。

Table 3.属性

名称	描述
base-package	定义查找所有继承*Repository的repository接口时所扫描包的路径，该路径下所有的包都会被扫描。允许使用通配符
repository-impl-postfix	定义自定义repository实现类的后缀。所有名字是以指定后缀结尾的类都会被看做是自定义repository的实现类。默认值是*Impl*
query-lookup-strategy	定义创建查询语句的策略，详细请看查询策略,默认采用*create-if-not-found*策略
named-queries-location	定义包含外部定义好的查询语句的Properties文件的位置
consider-nested-repositories	该属性的值决定了是否允许定义嵌套的repository接口。默认值是false

附录B Populators命名空间参考文档

元素

允许开发者通过Spring Data repository框架填充数据存储。

Table 4.Attributes(属性)

名称	描述
locations	定义包含用来读取填充对象的文件的位置

附录C Repository查询关键字

支持的查询关键字

下面的表格列出了Spring Data repository查询解析机制支持的查询关键字。某些特定的存储可能不支持全部的关键字。

Table 5.Query keywords(查询关键字)

逻辑关键字	关键字表达式
AND	And
OR	Or
AFTER	After, IsAfter
BEFORE	Before, IsBefore
CONTAINING	Containing, IsContaining, Contains
BETWEEN	Between, IsBetween
ENDING_WITH	EndingWith, IsEndingWith, EndsWith
EXISTS	Exists
FALSE	False, IsFalse
GREATER_THAN	GreaterThan, IsGreaterThan
GREATER_THAN_EQUALS	GreaterThanEqual, IsGreaterThanEqual
IN	In, IsIn
IS	Is, Equals, (or no keyword)
IS_NOT_NULL	NotNull, IsNotNull
IS_NULL	Null, IsNull
LESS_THAN	LessThan, IsLessThan
LESS_THAN_EQUAL	LessThanEqual, IsLessThanEqual
LIKE	Like, IsLike
NEAR	Near, IsNear
NOT	Not, IsNot
NOT_IN	NotIn, IsNotIn
NOT_LIKE	NotLike, IsNotLike
REGEX	Regex, MatchesRegex, Matches
STARTING_WITH	StartingWith, IsStartingWith, StartsWith
TRUE	True, IsTrue
WITHIN	Within, IsWithin

附录D Repository查询返回类型

支持的查询返回类型

下面的表格列出了Spring Data repositories支持的返回类型。某些特定的存储可能不支持全部的返回类型。

> 只有支持地理空间查询的数据存储才支持GeoResult，GeoResults，GeoPage等返回类型

Table 6.Query return types(查询返回值)

返回值类型	描述
void	表示没有返回值
Primitives	基本的Java类型
Wrapper types	Java的包装类
T	最多只返回一个实体。没有查询结果时返回null。如果超过了一个结果会抛出IncorrectResultSizeDataAccessException的异常
Iterator	一个迭代器
Collection	一个集合
List	一个列表
Optional	返回Java 8或Guava中的Optional类。查询方法的返回结果最多只能有一个。如果超过了一个结果会抛出IncorrectResultSizeDataAccessException的异常
Stream	Java 8引入的Stream类
Future	一个Future类，查询方法需要带有@Async注解，并开启Spring异步执行方法的功能
CompletableFuture	返回Java8中新引入的CompletableFuture类，查询方法需要带有@Async注解，并开启Spring异步执行方法的功能
ListenableFuture	返回org.springframework.util.concurrent.ListenableFuture类，查询方法需要带有@Async注解，并开启Spring异步执行方法的功能
Slice	返回指定大小的数据和是否还有可用数据的信息。需要方法带有Pageable类型的参数
Page	在Slice的基础上附加返回总数等信息。需要方法带有Pageable类型的参数
GeoResult	返回结果会附带诸如到相关地点距离等信息
GeoResults	返回GeoResult的列表，并附带到相关地点平均距离等信息
GeoPage	分页返回GeoResult，并附带到相关地点平均距离等信息